C++ Chess Game – Full Code Walkthrough

This document provides an **exhaustive, line-by-line tour** of the simple C++ console chess game contained in the project files you supplied (`main.cpp`, `game.cpp`, `board.cpp`, `piece.cpp`, `chess.h`, `build.bat`, `build.sh`). Explanations appear in the **chronological order in which the program executes**, beginning with `main.cpp` (the entry-point) and drilling down into the functions that are subsequently invoked. At every step we clarify C++ syntax that might be unfamiliar to a reader who is still learning the language.

Legend used in this document

Symbol	Meaning
▶	Statement executed at run-time in the shown order
`// …`	The original in-code comment from the author
italics	Commentary added by this document
bold	Key concept or important call-out

1. `main.cpp` (29 lines)

File purpose: program entry point, asks the user which mode to play and launches `ChessGame`.

Line	Code (trimmed)	Explanation
1-2	`#include "chess.h"` `#include <iostream>`	▶ The pre-processor copies the header file before compilation. `iostream` provides `std::cout` & `std::cin`.
4-5	`int main() {` `try {`	▶ Application starts here. A `try` block is opened so any C++ exception thrown inside can be caught and converted to an error message instead of crashing.
7-10	`std::string mode;` …	▶ Ask player to choose PvP vs PvCPU. `std::getline` reads an entire line including spaces.
11-17	`if (mode == "2") {` …	▶ When the user presses 2 a second prompt requests which **color** Stockfish (CPU) will play. A ternary operator converts the single-letter answer into the strongly-typed `enum class Color`. The constructor `ChessGame(true,cpuColor)` enables the engine.
18-22	`} else {` …	▶ Any other input starts a human-vs-human match. The constructor overload `ChessGame(false)` disables the CPU.
24-27	`} catch (const std::exception& e) {` …	▶ *Standard exception handling pattern in C++14*. The reference binds to the base-class of all STL exceptions so both `std::invalid_argument`, `std::runtime_error`, etc. are caught.

Line	Code (trimmed)	Explanation
28-29	`return 0;` `}`	▶ Returning zero indicates success to the host OS.

2. `chess.h` (177 logical lines)

Purpose: *single public header* that declares the **core domain classes** (`Position`, `Move`, every `Piece` subclass, `ChessBoard`, and `ChessGame`) plus a small wrapper helper for Stockfish.

• The file is wrapped in an **include-guard** (`#ifndef CHESS_H … #endif`), ensuring the compiler only includes these declarations once per translation unit.

Key Declarations (chronological order of use)

Code snippet	Explanation
`enum class Color { WHITE, BLACK };`	Strongly-typed enum (scoped) avoids implicit conversions to `int`, unlike a plain `enum`.
`struct Position { int row, col; … };`	Lightweight value type. Member functions implement *algebraic ↔ array index* conversion. `isValid()` bounds-checks coordinates. == operator is overloaded for easy comparison.
`struct Move { Position from,to; PieceType promotionPiece=PieceType::QUEEN; … };`	Bundles origin, destination and an *optional* promotion selection. Default member initialiser inside struct is a C++11 feature.
`class Piece { … virtual std::vector<Move> getPossibleMoves(const Position&,ChessBoard&) const =0; … };`	Abstract base class whose subclasses (Pawn, Rook, …) override polymorphic behaviours. The pure virtual (=0) method makes the class **abstract**.
`std::unique_ptr<Piece> board[8][8];`	Modern, RAII-safe ownership of dynamically allocated pieces; avoids manual `delete` and double-frees.
`void setEnPassant(const Position& pos);`	En-passant bookkeeping is stored in the board, **not** the pawn, which simplifies undo/redo logic.

(The remainder of the header merely declares the methods implemented in the .cpp files that follow, so their individual behaviour is discussed where it is first executed.)

3. `ChessGame` high-level execution path (`game.cpp`, 321 lines)

Once `main()` constructs `ChessGame`, the following chronological sequence occurs inside `ChessGame::startGame()`.

1. `std::cout << "Welcome …"` – printing banner.
2. `while (!gameOver)` loop – runs until checkmate, stalemate, or user quits.

3. `board.displayBoard()` – paints ASCII board (implemented later in `board.cpp`).
4. Branch depending on **who controls the side to move**.

3.1 CPU turn (when enabled)

Step	Relevant lines	Explanation
a	65-70	Builds a FEN string via `board.getFEN()` then calls `getBestMoveFromStockfish()` (line 130+) which spins up Stockfish as a **child process** on Windows using Win32 `CreateProcessA`.
b	80-92	After parsing the UCI reply (`bestmove e2e4`), converts it into `Move` and executes via `board.movePiece(move)`.
c	94-96	If move puts the enemy king in check an alert is printed.

3.2 Human turn

Lines 105-129 collect user input, trim whitespace, and delegate to `makeMove()`. That routine performs **full legality checking** by retrieving `board.getAllLegalMoves()` and rejecting moves not present.

> **Syntax highlight** – `std::remove` (line 113) returns an *erase-iterator*; the so-called *erase–remove idiom* requires a subsequent container `erase()` call, which is chained in a single expression here.

When a legal move is confirmed, `switchPlayer()` toggles `currentPlayer` and control returns to the top of the game loop.

`checkGameEnd()` finally determines check-, mate- or stalemate using helper methods in `ChessBoard`.

4. `board.cpp` (381 lines)

`ChessBoard` owns the 64 squares and implements **all move rules** as well as FEN generation and castling rights evaluation.

Below is a *timeline* of the calls made during a typical game, with references to their implementation lines.

Chronology	Function (line)	Key operations
1	`ChessBoard::ChessBoard()` (l.1-24)	Allocates empty 8×8 array and immediately calls `setupInitialPosition()`.
2	`setupInitialPosition()` (l.65-108)	Builds the classical start diagram by creating each `Piece` subclass with `std::make_unique`. *Syntax: template argument deduction introduced in C++14 removes the need to repeat the type.*

Chronology	Function (line)	Key operations
3	`displayBoard()` (l.110-139)	Renders from rank 8 down to 1 with file labels beneath. Uses `std::cout << '\n'` to end lines rather than `std::endl` for **performance** (no flush needed).
4	`movePiece(const Move&)` (l.155-230)	The engine-core; performs *five* special-case checks in order:
	• en-passant capture	
	• castling	
	• pawn promotion	
	• sets hasMoved flag on first move of king/rook/pawn	
	• updates `enPassantTarget` if a pawn double-steps	
5	`isSquareAttacked()` (l.260-315)	Determines whether a square is under attack **without recursion loops** by temporarily casting away const-ness to call polymorphic `getPossibleMoves()`.
6	`getAllLegalMoves()` (l.340-364)	Iterates over every own piece and filters out moves that leave the king in check by simulating them on a *copy* of the board (`ChessBoard tempBoard = *this;`).
7	`isCheckmate()`/`isStalemate()` (l.370-376) wrap `isInCheck()` + `getAllLegalMoves().empty()`.	
8	`getFEN()` (l.380-406)	Serialises the full position but omits castling + half-move counters for simplicity, returning a valid though minimal FEN understood by Stockfish.

5. `piece.cpp` (221 lines)

Contains concrete movement logic for each chess piece. To avoid duplication the file introduces `addDirectionalMoves()` – a helper that walks in a straight or diagonal line and collects legal squares until blocked, used by *rook, bishop, and queen*.

5.1 Utility functions

```
int direction = (color == Color::WHITE) ? 1 : -1;
```

Conditional operator returns `1` for white (which moves upward in our 0-indexed array) and `-1` for black.

5.2 Class-specific rules

Class	Notable rule implementation
`Pawn`	double-step from rank 2/7, diagonal capture, promotion row check, **en-passant** target detection via `board.isEnPassantTarget()`.
`Knight`	Hard-coded **L-shaped offsets**; always ignores intervening pieces.
`Bishop`	Calls `addDirectionalMoves()` with the four diagonal vectors.
`Rook`	Same helper with rank/file vectors.
`Queen`	Combination of rook + bishop vectors in a single 8-direction table.
`King`	Single-square moves plus *castling* by testing `board.canCastle()`; promotion of rook handled back in `movePiece()`.

6. Build scripts

`build.bat` (Windows)

Simple batch wrapper that compiles with g++ (MinGW) and prints a success/failure banner. `%errorlevel%` contains the exit status of the last command; the `if` statement is a Windows cmd shell conditional.

`build.sh` (Unix/macOS/Linux)

Bash counterpart. `$?` is the most recent command status; `exit 1` propagates error to calling shell/CI.

Both scripts pass the same flags: `-std=c++14 -Wall -Wextra -O2` meaning *C++14 standard*, *all warnings*, *extra warnings*, and `-O2` optimisation.

7. Key C++ 14 Syntax Highlights

1. **Smart pointers** – `std::unique_ptr<T>` ensures single-ownership semantics and automatic destruction, replacing raw `new`/`delete`.
2. **Range-based for loops** (e.g., `for (auto& dir : directions)`) simplify iteration without explicit indices.
3. **Uniform initialisation** `{ }` is used in enum tables and struct members, enforcing type safety.
4. **Override specifier** – `getPossibleMoves(...) const override` guarantees the base class actually declared a virtual of the same signature.
5. **Enum class** gives strongly typed enumerations avoiding accidental arithmetic.
6. `static_cast` **vs** `reinterpret_cast` – none of the dangerous casts appear; pointer conversions rely on polymorphism instead.

Final one-paragraph summary

This compact C14 console chess engine starts execution in `main.cpp`, queries the user for game mode, and then delegates to the `ChessGame` object, which supervises an endless loop of alternating turns until `checkGameEnd()` signals mate, stalemate, or resignation. `ChessGame` drives a `ChessBoard` instance that maintains the 8×8 grid via `std::unique_ptr` and enforces all rules—movement generation in `piece.cpp`, legality filtering (self-check) in `board.cpp`, and optional Stockfish integration for AI via a child process. Clean object-oriented design, RAII memory management, and modern C14 language features such as scoped enumerations, smart pointers, and range-based loops make the code robust and succinct while still readable for learners.